Cache Memories

Topics
- Generic cache memory organization
- Direct mapped caches
- Set associative caches
- Impact of caches on performance

Next time
- Linking
Cache memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, then in main memory.
- Typical bus structure:
Measuring Cache Effects

- Memory mountain test code
  - Measures read throughput as a function of spatial and temporal locality.
  - Read throughput (read bandwidth) = Number of bytes read from memory per second (MB/s)
  - Graph throughput over changes in stride and working set size (number of repeatedly referenced locations)
  - Compact way to characterize memory system performance.
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */

    Mhz = mhz(0);              /* Estimate the clock frequency */

    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));

        printf("\n");
    }

    exit(0);
}
Memory mountain test function

```c
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                      /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```
The memory mountain

Pentium III Xeon
550 MHz
16 KB on-chip L1 d-cache
16 KB on-chip L1 i-cache
512 KB off-chip unified
L2 cache

Slopes of 
Spatial 
Locality

Ridges of 
Temporal 
Locality

stride (words)

working set size (bytes)

EECS 213 Introduction to Computer Systems
Northwestern University

Monday, November 7, 2011
Ridges of temporal locality

- Slice through the memory mountain with stride=1
  - illuminates read throughputs of different caches and memory
A slope of spatial locality

- Slice through memory mountain with size=256KB
  - shows cache block size.

![Graph showing read throughput vs stride for different strides](image)
Direct-mapped cache

- Simplest kind of cache
- Cache divided in $S$ sets of $N$-byte blocks
  - $N = 2^b$, $S = 2^s$
  - Typically, $N = 32$ or $64$ (our examples use 4 bytes)
  - Blocks capture spatial locality
- Valid bit = 1 if data in stored in set $i$
- Tag field identifies which address is currently stored

\[
\begin{array}{ccc}
\text{set 0:} & \text{valid} & \text{tag} & \text{cache block} \\
\text{set 1:} & \text{valid} & \text{tag} & \text{cache block} \\
\cdots \\
\text{set S-1:} & \text{valid} & \text{tag} & \text{cache block}
\end{array}
\]
Accessing direct-mapped caches

- Low $b$ bits determine block offset
- Middle $s$ bits of address determine index set
- Store remaining $t$ bits in tag
Accessing direct-mapped caches

Example: 16 bit addresses, 8 sets, 8 byte block in each set

$\text{To store} \quad 1011\ 0011\ 0101\ 1101$

$\begin{array}{c}
000 \\
001 \\
010 \\
011 \\
100 \\
101 \\
110 \\
111 \\
\end{array}$

$\begin{array}{c}
\text{tag}=10 \\
\text{s}=3 \\
\text{b}=3 \\
\end{array}$

$\begin{array}{c}
\text{xxxxxxxxxxx} \\
\text{xxx} \\
\text{xxx} \\
\end{array}$

$1011001101$ $011$ $101$
Checkpoint
Why use middle bits as index?

- High-order bit indexing
  - Adjacent memory lines would map to same cache entry
  - Spatially local code would have more cache conflicts

- Middle-order bit indexing
  - Consecutive memory lines map to different cache lines
  - Can hold C-byte region of address space in cache at one time
Set associative caches

- Characterized by more than one line per set

![Diagram of set associative caches]

- Set 0:
  - Valid
  - Tag
  - Cache block

- Set 1:
  - Valid
  - Tag
  - Cache block

- Set S-1:
  - Valid
  - Tag
  - Cache block

\( E=2 \) lines per set
Accessing direct-mapped caches

- **Line matching**
  - Find a valid line in the selected set with a matching tag
- **Word selection**
  - Then extract the word

```
0110
```

1. The valid bit must be set
2. The tag bits in the cache line must match the tag bits in the address
3. If (1) and (2), then cache hit, and block offset selects starting byte.
General org of a cache memory

Cache is an array of sets.

Each set contains one or more lines.

Each line holds a block of data.

$S = 2^s$ sets

Cache size: $C = S \times E \times B$ data bytes

$E$ lines per set

$B = 2^b$ bytes per cache block

$1$ valid bit per line

$t$ tag bits per line

Set 0:

valid | tag | 0 1 \cdots B-1

Set 1:

valid | tag | 0 1 \cdots B-1

Set S-1:

valid | tag | 0 1 \cdots B-1
Accessing set associative caches

- Set selection
  - identical to direct-mapped cache
Accessing set associative caches

- Line matching and word selection
  - must compare the tag in each valid line in the selected set.

10110

\[ w_3 \]
\[ w_2 \]
\[ w_1 \]
\[ w_0 \]

1001

\[ t \] \[ s \] \[ b \]

0110

\[ t \text{ bits} \]
\[ s \text{ bits} \]
\[ b \text{ bits} \]

\[ 0110 \]
\[ i \]
\[ 100 \]

\( m^{-1} \text{ tag} \)
\( \text{set index} \)
\( \text{block offset}^0 \)

=1?  (1) The valid bit must be set.

= ?  (2) The tag bits in one of the cache lines must match the tag bits in the address

(3) If (1) and (2), then cache hit, and block offset selects starting byte.
Addressing caches

Address A:

\[
\begin{array}{c|c|c|c}
\text{t bits} & \text{s bits} & \text{b bits} \\
\hline
m-1 & & 0 \\
\end{array}
\]

\(<\text{tag}>\) \hspace{0.5cm} \(<\text{set index}>\) \hspace{0.5cm} \(<\text{block offset}>\)

The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.
Cache Parameters

- $S = 2^s$: number of sets
- $E$: number of lines / set ($E = 1$ direct-mapped)
- $B = 2^b$: block size in bytes
- $m = \log_2(M)$: number of address bits
- $t = m - (s + b)$: number of tag bits
- $C = B \times E \times S$: cache size in bytes (blocks only, not valid and tag bits)
Checkpoint
Multi-level caches

- Options: separate data and instruction caches, or a unified cache

<table>
<thead>
<tr>
<th></th>
<th>Regs</th>
<th>L1 d-cache</th>
<th>L1 i-cache</th>
<th>Unified L2 Cache</th>
<th>Memory</th>
<th>disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:</td>
<td>200 B</td>
<td>8-64 KB</td>
<td>1-4MB SRAM</td>
<td>128 MB DRAM</td>
<td>30 GB</td>
<td></td>
</tr>
<tr>
<td>speed:</td>
<td>3 ns</td>
<td>3 ns</td>
<td>6 ns</td>
<td>60 ns</td>
<td>8 ms</td>
<td></td>
</tr>
<tr>
<td>$/Mbyte:</td>
<td>$100/MB</td>
<td>$1.50/MB</td>
<td>$100/MB</td>
<td>$1.50/MB</td>
<td>$0.05/MB</td>
<td></td>
</tr>
<tr>
<td>line size:</td>
<td>8 B</td>
<td>32 B</td>
<td>32 B</td>
<td>8 KB</td>
<td>8 KB</td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper
Intel Pentium Cache Hierarchy

Processor Chip

L1 Data
1 cycle latency
16 KB
4-way assoc
Write-through
32B lines

L2 Unified
128KB--2 MB
4-way assoc
Write-back
Write allocate
32B lines

Main Memory
Up to 4GB

L1 Instruction
16 KB, 4-way
32B lines

Regs.
Cache performance metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses/references)
  - Typical numbers:
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
  - Typical numbers:
    - 1 clock cycle for L1
    - 3-8 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - Typically 25-100 cycles for main memory
Writing cache friendly code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - assume cold cache, 4-byte words, 4-word cache blocks

```c
int sumarrayrows(int a[M][N]) {
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```c
int sumarraycols(int a[M][N]) {
    int i, j, sum = 0;
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%
Matrix multiplication example

- Major cache effects to consider
  - Total cache size
    - Exploit temporal locality and keep the working set small (e.g., by using blocking)
  - Block size
    - Exploit spatial locality

- Description:
  - Multiply N x N matrices
  - O(N³) total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```c
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` held in register
Miss rate analysis for matrix multiply

• Assume:
  – Line size = 32B (big enough for 4 64-bit words)
  – Matrix dimension (N) is very large
    • Approximate 1/N as 0.0
  – Cache is not even big enough to hold multiple rows

• Analysis method:
  – Look at access pattern of inner loop

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C}
\end{array}
\]

Monday, November 7, 2011
Layout of C arrays in memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations

- Stepping through columns in one row:
  - for (i = 0; i < N; i++)
    sum += A[0][i];
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    • compulsory miss rate = 4 bytes / B

- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    sum += A[i][0];
  - accesses distant elements
  - no spatial locality!
    • compulsory miss rate = 1 (i.e. 100%)
Conflict misses in Direct-Mapped Caches

```c
float dotprod(float x[8], float y[8])
{
    float sum = 0.0; int i;
    for (i = 0; i < 8; i++)
        sum += x[i] * y[i];
    return sum;
}
```

- Assume for simplicity
  - 4-byte floats
  - x[] loaded at address 0, y[] at address 32
  - 16 byte cache block (4 floats)
  - 2 sets (cache size = 32 bytes)

- x[0] – x[3] and y[0] – y[3] map to set 0
- Almost every array reference clobbers the same cache set
- This is called thrashing. Can make code 2 or 3 times slower.
- Fix by padding arrays to avoid powers of 2, e.g., x[12] and y[12].
Matrix multiplication (ijk)

```c
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```

- **Misses per inner loop iteration:**
  - A: 0.25
  - B: 1.0
  - C: 0.0

Inner loop:
- Row-wise
- Column-wise
- Fixed
Matrix multiplication (jik)

```c
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] = sum
  }
}
```

- **Misses per inner loop iteration:**
  - A: 0.25
  - B: 1.0
  - C: 0.0
Matrix multiplication (jki)

```c
/* jki */
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = B[k][j];
        for (i=0; i<n; i++)
            C[i][j] += A[i][k] * r;
    }
}
```

**Misses per inner loop iteration:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misses per iteration</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Monday, November 7, 2011
Matrix multiplication (kji)

```c
/* kji */
for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
        r = B[k][j];
        for (i=0; i<n; i++)
            C[i][j] += A[i][k] * r;
    }
}
```

• Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Matrix multiplication (kij)

```c
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = A[i][k];
    for (j=0; j<n; j++)
      C[i][j] += r * B[k][j];
  }
}
```

- **Misses per inner loop iteration:**
  - A: 0.0
  - B: 0.25
  - C: 0.25

**Inner loop:**
- Fixed
- Row-wise
- Row-wise
Matrix multiplication (ikj)

/* ikj */
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        r = A[i][k];
        for (j=0; j<n; j++)
            C[i][j] += r * B[k][j];
    }
}

• Misses per inner loop iteration:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0</td>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>
# Summary of matrix multiplication

ijk & jik:
- 2 loads, 0 stores
- misses/iter = 1.25

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```

jki & kji:
- 2 loads, 1 store
- misses/iter = 2.0

```c
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        r = B[k][j];
        for (i=0; i<n; i++)
            C[i][j] += A[i][k] * r;
    }
}
```

kij & ikj:
- 2 loads, 1 store
- misses/iter = 0.5

```c
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = B[k][j];
        for (j=0; j<n; j++)
            C[i][j] += A[i][k] * r;
    }
}
```

jki & kji:
- 2 loads, 1 store
- misses/iter = 2.0

```c
for (k=0; k<n; k++) {
    for (i=0; i<n; i++) {
        r = A[i][k];
        for (j=0; j<n; j++)
            C[i][j] += r * B[k][j];
    }
}
```
Miss rates are helpful but not perfect predictors.

Code scheduling matters, too.
Improving temporal locality by blocking

- Example: Blocked matrix multiplication
  - “block” (in this context) does not mean “cache block”.
  - Instead, it mean a sub-block within the matrix.
  - Example: $N = 8$; sub-block size = 4

\[
\begin{bmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
  B_{11} & B_{12} \\
  B_{21} & B_{22}
\end{bmatrix}
=
\begin{bmatrix}
  C_{11} & C_{12} \\
  C_{21} & C_{22}
\end{bmatrix}
\]

**Key idea:** Sub-blocks (i.e., $A_{xy}$) can be treated just like scalars.

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad C_{12} = A_{11}B_{12} + A_{12}B_{22}
\]
\[
C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]
Blocked matrix multiply (bijk)

```
for (jj=0; jj<n; jj+=bsize) {
    for (i=0; i<n; i++)
        for (j=jj; j < min(jj+bsize,n); j++)
            c[i][j] = 0.0;
    for (kk=0; kk<n; kk+=bsize) {
        for (i=0; i<n; i++)
            for (j=jj; j < min(jj+bsize,n); j++)
                sum = 0.0
                for (k=kk; k < min(kk+bsize,n); k++)
                    sum += a[i][k] * b[k][j];
                c[i][j] += sum;
    }
}
```
Blocked matrix multiply analysis

- Innermost loop pair multiplies a $1 \times bsize$ sliver of $A$ by a $bsize \times bsize$ block of $B$ and accumulates into $1 \times bsize$ sliver of $C$
- Loop over $i$ steps through $n$ row slivers of $A$ & $C$, using same $B$

```c
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
    }
}
```

Update successive elements of sliver
Pentium blocked matrix mult performance

- Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)
  - relatively insensitive to array size.
Concluding observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- All systems favor “cache friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)